FORTINET
High Performance Network Security

FortiWLC

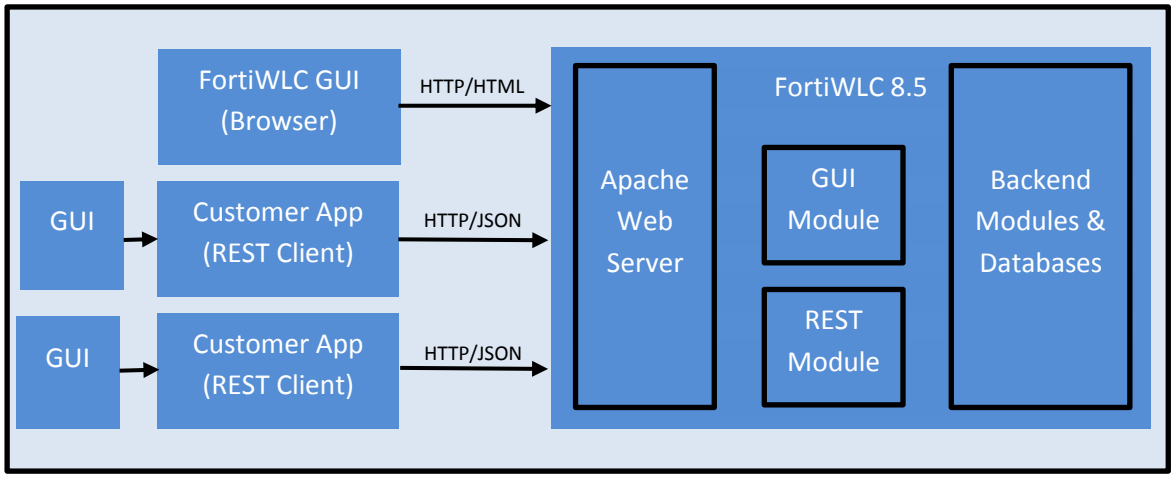
REST API Reference Guide

Release 8.5.0

Table of Contents

REST API Scope

Both configuration and monitoring are supported over REST API. Future versions may support notifications and diagnostics as well.



Request and Response Data Format

JSON (specify 'Content-Type': 'application/json' in header)

Sample Client Programs

The sample client programs are available in following programming languages:

- **Python** (2.7.9+) - requires *requests* and *responses* external packages
- ***Java*** *(SE 7+) – (future release)*
- ***C#*** *- (future release)*

One can also use *curl* command as below. Note that Basic Authentication is used by supplying username and password as part of the URL.

$ curl "https://*username*:*password*@***host***/api/v10/cfg/devices/controller/controller/" -k

One can also use regular web browsers:

https://***host***/api/v10/cfg/devices/controller/controller/

OR

https://*username*:*password*@***host***/api/v10/cfg/devices/controller/controller/

In former case where user credentials are not supplied, the browser prompts for them in a login dialog box.

REST and Endpoint URLs

REST (**RE**presentational **S**tate **T**ransfer) is a modern, scalable (but not high performance) client-server based RPC technique using existing HTTP protocol methods (such as GET, POST, PUT, DELETE) on server resources (identified by URLs) and transferring the resources in either XML / JSON / HTML representation.

Talking about resources on the controller, there are two kinds: *forms* and *tables*.

Forms are data-structures with one instance and instances are already created and cannot be deleted; they can only be retrieved and updated (modified). This means, only GET (query) and PUT (update) methods work with forms. Each form is uniquely identifies by its name.

Tables are like collections (a list of data-structures). These tables are already created and cannot be deleted as a whole. However, tables can be queried, and individual entries can be added / modified / deleted / queried. This means, only GET (query) and POST (add) methods work with table collection while GET (query), PUT (update) and DELETE (remove) methods work with an entry in the table. Each table is uniquely identified by its name and entries in it are uniquely identified by its primary-key (such as profile name) and in some cases additionally by other table parameters.

API for both forms and tables are offered to REST clients as their own URLs. For example, API for a *form* called "*system_resource*" can be offered as URL:

/api/v10/monitor/devices/system_resources

Since, multiple methods are allowed on the URL (in this case GET and PUT methods since it is a form) and each method on the URL may require different user privilege level to carry out the method, the REST module maintains a hash-table of URLs to form names per method.

GET["/api/v10/monitor/devices/system_resources"] **=>** FORM, SYSTEM_RESOURCE, PL1

PUT["/api/v10/monitor/devices/system_resources"] **=>** FORM, SYSTEM_RESOURCE, PL2

The URL, as can be seen above, maps in REST module to a form called "*system_resource"* in two hash-tables: GET hash-table and PUT hash-table (here, PL means minimum required user *privilege level* to execute the method which will be used in *authorization* discussed later). Thus, when GET is executed on this URL, the configured form is retrieved and when PUT is executed on this URL, the configured form is updated. Other methods return UNSUPPORTED_METHOD for the forms.

Similarly, a *table* URL and its hash-tables to table name in REST module look as below:

GET["/api/v10/cfg/security/security_profiles"] => TABLE, TABLE_SECURITY_PROFILE, PL

POST["/api/v10/cfg/security/security_profiles"] => TABLE, TABLE_SECURITY_PROFILE, PL

GET["/api/v10/cfg/security/security_profiles/$"] => TABLE, TABLE_SECURITY_PROFILE, PL, "Name"

PUT["/api/v10/cfg/security/security_profiles/$"] => TABLE, TABLE_SECURITY_PROFILE, PL, "Name"

DEL["/api/v10/cfg/security/security_profiles/$"] => TABLE, TABLE_SECURITY_PROFILE, PL, "Name"

The URL maps in REST module to a table called "*security_profile"* in 5 hash-tables. The first two pertain to the table (collection) itself (query table and add new entry to table) where the base URL is used, whereas rest three pertain to single entry in the table where the URL also contains a dollar sign to indicate that one variable is expected in the URL after the base URL and that maps to "*Name*" attribute in the table. Usually, this is the primary-key of the table that identifies entries uniquely.

Thus, REST module maintains a list of URL to form or table name mapping per method. When a new feature needs to be supported, its public URL and internal form or table name needs to be added to these hash-tables in the REST module.

Forms	
GET /resource	Retrieves a specified form. The URL internally maps to a form name. Request contains no JSON data as URL identifies the form to retrieve. Response contains a JSON object containing form data. Example: GET /api/v10/monitor/devices/system_resources
PUT /resource	Updates a specified form. The URL internally maps to a form name. Request contains a JSON object with one or more form parameters. Response contains a JSON object containing result.
Table (collection)	
Entire Collection	
GET /resources	Retrieves all entries in specified table (max N entries). The URL internally maps to a table name. Request contains no JSON data as URL identifies the table to retrieve. Response contains JSON **list** of objects (table entries). Note that, this retrieves a list of objects. Example: GET /api/v10/cfg/security/security_profiles This maps to *security_profile* with type as TABLE in REST module.
POST /resources	Adds a new entry to specified table. The URL internally maps to a table name. Request contains a JSON object containing a new table entry. Response contains a JSON object containing result.
Named Entry in a Collection	
GET /resources/id PUT /resources/id DEL /resources/id	Retrieves / Updates / Deletes an entry in specified table. The entry is identified by a parameter after the collections URL. Example: GET /api/v10/cfg/security/security_profiles/test This maps to *security_profile* with type as TABLE in REST module and one resolved parameter as *test*.

Client (REST) Side Authentication & Authorization

The REST module currently offers Basic authentication and future versions may support Bearer/Token Authentication and Sessions as well.

- **Basic Authentication:** Here, the user supplies credentials (request header ***Authorization*: *Basic***) in each request. That is, this type of authentication is *stateless*! Thus, no *login* and *logout* calls are used here, enabling one to execute HTTP methods on resources requiring credentials but without overhead of logging in.

- **Token Authentication (future release):** Here, the user performs login request with his credentials (username and password), and the REST module validates the user and returns an *access token* in the JSON response. That is, the JSON response contains *access_token* property. The client supplies the same access token in subsequent requests (using request header ***Authorization*: *Bearer***) and the REST module decodes the token to identify the user. Since access tokens independently identify the user login without need to store them into a session, the **Token Authentication** also works in stateless mode. Thus, *logout* request has no effect on Token Authentication.

Users can be *admin*, locally configured users as well as RADIUS and TACACS+ users.

Both Basic Authentication and Bearer Authentication are more scalable since each request can be serviced independently whereas in sessions, the session context needs to be shared (via an external database) with other servers or all requests of a session must be sent to same server.

Session (future release): User will perform a *login* request by supplying his credentials (username and password). The REST module validates the user and session identifier is internally encrypted into a session *cookie*. Sessions are removed in a call to *logout* (or inactivity *timeout*). After a successful call to login, subsequent calls in the session do not carry user credentials but just a session identifier in the form of a cookie.

Following is how this *cookie* based sessions work. When a client makes *login* request, the REST module creates a new session object that holds an internally generated unique session-id, logged-in username and his privilege level, and includes the session identifier in a cookie in the response (response header ***Set-Cookie*** with session-id and no expiry date). The JSON response also contains the *session_id* property. The client needs to send this session cookie (request header ***Cookie***) in subsequent requests until and including *logout* request when the session will be removed and cleared cookie (response header ***Set-Cookie*** with empty value and expired date) is sent in the response so that client too removes the cookie from its store.

Note: When sessions will be implemented in future release, either Sessions or Token Authentication can be enabled at a time because both involve *login* request. By default Token Authentication will be enabled since it is stateless and hence REST module does not need to store the sessions and manage them.

When basic/bearer authentication is missing in a request, the module responds with HTTP_UNAUTHORIZED with header **WWW-Authenticate**: **Basic realm="full-wlc-rest-api"** so that the client can repeat the request using basic authentication mechanism.

To avoid session hijacks or reveal trivial basic authentication details used, the REST module must be used under a secure connection (https).

Authorization: Authorization means applying requesting user privilege level to requested resources (URLs). In REST module, each URL + Method combination has its associated minimum required privilege level to execute it. For example, **GET *url1*** might have minimum required privilege level of 5 while POST on same resource might have minimum required privilege level of 12. This configuration is hardcoded in the source code (not configurable presently).

Similarly, each user has his associated privilege level. Thus, when a user is attempting to execute a method on a URL, the REST module performs *authorization* by comparing user's actual privilege level with the minimum required privilege level configured on that URL + Method combination. The request succeeds only if user's privilege level is more than or equal to the minimum required privilege level on the requested resource using requested method. In other words, if required privilege level of a resource+method is more than user's actual privilege level, then request fails.

Note: Few resources (**/api/v10/version/** and **/api/v10/schema/**) do *not* require authentication and authorization and hence can always be executed.

Basic API URNs

These are some general API URNs.

Purpose	URN	Methods
Version	/api/v10/version/	GET
Schema	/api/v10/schema/	GET
Login	/api/v10/login/	POST
Logout	/api/v10/logout/	GET | POST | PUT

Login (HTTP POST)

When using basic authentication, no login is required – just provide username and password in every request using request header ***Authorization*: *Basic***!

However, when using bearer/token authentication (future release), login call will be mandatory and must be the first call before any subsequent meaningful call can be made. There are quite a few calls that can be made without login such as querying version number. One needs to provide his/her username and password during login call (in JSON format as shown below – parameters are case-sensitive). The response is also in JSON format. Upon authentication failure, proper error response is returned (for instance, user does not exist). Upon success, an access-token is created and returned in response. This token must be sent in all subsequent calls - using request header ***Authorization*: *Bearer***. No logout call is required at the end.

Login HTTP POST Request Body	Login HTTP 200 OK Response Body

{
"username": "*$username* ",
"password": "*$password* "
}
	Failure response (example):
	{
	"_meta" : {
	"Status": "Failure",
	"Message": "608 => incorrect login"
	}
	}
	
	Success response:
	{
	"_meta": {
	"Status": "Success"
	}
	"_items": {
	"username": "*$username* ",
	"privilege_level": "*$privilege-level*",
	"access_token": "YWRtaW46YWRtaW4="
	"expiry_time": "*$expirytime*",
	}
	}

User privilege-level is a number between 1 (lowest) and 15 (highest). Any user can query configurations but only users with privilege-level of 10 and above can set configurations.

General format of HTTP Request Body and Response Body

Format of JSON Request Body (POST | PUT | DELETE)	Format of JSON Response Body

Ex: create or update a security profile:
 	
{
"Name": "test,
"L2ModesAllowed": "1000000",
"CypherSuites": "10000",
"AuthenticationSuites": " 1",
"PskKey": "12341234",
"CaptivePortalEnabled": "disabled"
}
	Error response:
	{
	"_meta" : {
	"Status": "Failure",
	"Message": "*description*"
	}
	}
	
	Querying a collection (table):
	{
	"_meta" : {
	"Status": "Success",
	"StartRecord": X,
	"EndRecord": Y,
	"TotalRecords": Z
	}
	"_items": [
	{
	},
	{
	}
	]
	}
	
	Querying a single object (form or an entry in a table):
	{
	"_meta" : {
	"Status": "Success"
	}
	
	"_items": {
	}
	}
	
	Create / Modify / Delete objects:
	{
	"_meta" : {
	"Status": "Success"
	}
	}

Schema (data syntax)

Each object-type has its own schema that depends on controller version. To get schema for all objects:

GET https://***wlc-host***/api/v10/schema

Pagination - Tabular Data Navigation

While querying a collection (table), filter parameters (**startrecord** and **endrecord**, index 1 based) may be supplied. Maximum range is 500. Default values are 1 and 500.

API URN DETAILS

Many requests require just the URL while some require parameters to URLs. For example, querying all security profiles just requires a base URL while querying a particular security profile requires a parameter (profile name) appended to the base URL.

Examples:

$ curl -X GET https://user:pass@***host***/api/v10/cfg/security/security_profiles/ -k

$ curl -X GET https://user:pass@***host***/api/v10/cfg/security/security_profiles***/default*** -k

Here, the first one queries all the available security profiles while second one queries a security profile with name ***default***. If the requested profile does not exist, then an error is returned, otherwise security profile contents are returned.

Apart from parameters in the URL, the PUT HTTP method require (JSON) data in their request body. For example, updating/modifying a security profile requires security profile name in the URL and also the new content in the request body (JSON format).

All responses are in JSON.

The REST API follows modularity found in controller GUI. To get full list of API URLs and their description and schema:

GET https://***wlc-host***/api/v10/schema

As described above, developers can retrieve schema (object data syntax) associated with each API URL and then construct JSON request and/or parse JSON response accordingly.

Appendix

API URNs & Schema (generated html file)

For API URNs and schema, see file ***FortiWLC REST API – Schema.rar***.

Client Sample (Python)

For a sample REST client program, see file ***wlc_rest_client_sample.py***.

Client Sample (SecuredPSK)

For a sample REST client program for SecuredPSK, see file ***wlc_rest_client_sample_spsk.py***